

Safe Integer Operations

Daniel Plakosh, Software Engineering Institute [vita¹]

Copyright © 2005 Pearson Education, Inc.

2005-09-27

Integer operations can result in error conditions and lost data, particularly when inputs to these operations can be manipulated by a malicious user. A solution to this problem is to use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source.

Development Context

Integer operations

Technology Context

C, C++, IA-32, Win32, UNIX

Attacks

Attacker executes arbitrary code on machine with permissions of compromised process or changes the behavior of the program.

Risk

Integers in C and C++ are susceptible to overflow, sign, and truncation errors that can lead to exploitable vulnerabilities.

Description

Integer operations can result in error conditions and lost data, particularly when inputs to these operations can be manipulated by a malicious user.

The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program.

An alternative or ancillary approach is to protect each operation. However, because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labor intensive and expensive to implement.

A more economical solution to this problem is to use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source. Figure 1 shows examples of

1. daisy:268 (Plakosh, Daniel)

when to use safe integer operations.

Figure 1. Checking for overflow when adding two signed integers

Use Safe Integer Operations

```
void* CreateStructs(int StructSize, int HowMany) {
    SafeInt<unsigned long> s(StructSize);
    s *= HowMany;
    return malloc(s);
}
```

Don't Use Safe Integer Operations

```
void foo() {
    int i;
    for (i = 0; i < INT_MAX; i++)
        ....
}
```

The first example shows a function that accepts two parameters specifying the size of a given structure and the number of structures to allocate that can be manipulated by untrusted sources. These two values are then multiplied to determine what size memory to allocate. Of course, the multiplication operation could easily overflow the integer variable and provide an opportunity to exploit a buffer overflow.

The second example shows when *not* to use safe integer operations. The integer *i* is used in a tightly controlled loop and is not subject to manipulation by an untrusted source, so using safe integers would add unnecessary performance overhead.

Safe integer libraries use different implementation strategies. The gcc library uses postconditions to detect integer errors. SafeInt C++ class tests preconditions to prevent integer errors. The Michael Howard library takes advantage of machine-specific mechanisms to detect integer errors. We compare and contrast these approaches in the remainder of this section.

GCC

The gcc runtime system generates traps for signed overflow on addition, subtraction, and multiplication operations for programs compiled with the `-ftrapv` flag. To accomplish this, calls are made to existing, portable library functions that test an operation's postconditions and call the C library `abort()` function when results indicate that an integer error has occurred.

Figure 2. Checking for overflow when adding two signed integers

```
1. Wtype __addvsi3 (Wtype a, Wtype b) {
2.     const Wtype w = a + b;
3.     if (b >= 0 ? w < a : w > a)
4.         abort();
5.     return w;
6. }
```

Figure 2 shows a function from the gcc runtime system that is used to detect overflows resulting from the addition of signed 16-bit integers. The addition operation is performed on line 2 and the results are compared to the operands to determine whether an overflow condition has occurred. For `__addvsi3()`, if *b* is non-negative and *w* < *a*, an overflow has occurred and `abort()` is called. Similarly, `abort()` is also called if *b* is negative and *w* > *a*.

C Language Compatible Library

Michael Howard has written parts of a safe integer library that detects integer overflow conditions using architecture-specific mechanisms [Howard 03b].

Figure 3. Unsigned integer addition and multiplication operations

```
1. in bool UAdd(size_t a, size_t b, size_t *r) {
2.     __asm {
3.         mov eax, dword ptr [a]
4.         add eax, dword ptr [b]
5.         mov ecx, dword ptr [r]
6.         mov dword ptr [ecx], eax
7.         jc short j1
8.         mov al, 1 // 1 is success
9.         jmp short j2
10.    j1:
11.        xor al, al // 0 is failure
12.    j2:
13.    };
14. }
```

Figure 3 shows a function that performs unsigned addition. Figure 5 shows a version of the vulnerable program from Figure 4 that has been modified (shown in bold) to use the Howard library. The calculation of the total length of the two strings is performed using the `UAdd()` call on lines 3–4 with appropriate checks for error conditions. Even adding one to the sum can result in an overflow and needs to be protected.

The Howard approach can be used in both C and C++ programs. However, the API is awkward to use and portability to other hardware architectures is lost due to the use of embedded Intel assembly instructions. Ironically, the use of embedded Intel assembly instructions does not necessarily improve the performance of the final assemblies because these instructions can impede the compiler’s ability to generate fully optimized code.

Figure 4. Truncation error involving the sum of two lengths

```
1. int main(int argc, char *const *argv) {
2.     unsigned short int total;
3.     total = strlen(argv[1])+strlen(argv[2])+1;
4.     char *buff = (char *) malloc(total);
5.     strcpy(buff, argv[1]);
6.     strcat(buff, argv[2]);
7. }
```

Figure 5. C language compatible library solution

```
1. int main(int argc, char *const *argv) {
2.     unsigned int total;
3.     if (UAdd(strlen(argv[1]), 1, &total) && UAdd(total, strlen(argv[2]), &total)) {
4.         char *buff = (char *)malloc(total);
5.         strcpy(buff, argv[1]);
6.         strcat(buff, argv[2]);
7.     }
8.     else {
9.         abort();
10.    }
11. }
```

SafeInt Class

SafeInt is a C++ template class written by David LeBlanc [LeBlanc 04]. SafeInt generally implements the precondition approach and tests the values of operands before performing an operation to determine whether errors might occur. The class is declared as a template, so it can be used with any integer type. Nearly every relevant operator has been overridden except for the subscripting operator `[]`.

Figure 6. Checking for overflow when adding two signed integers

```
1. if (!(rhs ^ lhs) < 0) { //test for +/- combo
2.     //either two negatives or two positives
3.     if (rhs < 0) {
4.         //two negatives
5.         if (lhs < MinInt() - rhs) { //remember rhs < 0
6.             throw ERROR_ARITHMETIC_UNDERFLOW;
7.         }
8.         //ok
9.     }
10.    else {
11.        //two positives
12.        if (MaxInt() - lhs < rhs) {
13.            throw ERROR_ARITHMETIC_OVERFLOW;
14.        }
15.        //OK
16.    }
17. }
18. //else overflow not possible
19. return lhs + rhs;
```

Figure 6 shows a section of code from the SafeInt class that checks for overflow in signed integer addition. Figure 7 shows a version of the vulnerable program from Figure 4 that has been modified (boldface type) to use the SafeInt library. Lines 1–4 show the implementation for the SafeInt + operator, which is invoked twice on line 9 of the main routine. The variables s1 and s2 are declared as SafeInt types on lines 7 and 8. In both cases, the SafeInt class is instantiated as an unsigned long type. When the + operator is invoked (twice) on line 9, it uses the safe version of the operator implemented as part of the SafeInt class. The safe version of the operator guarantees that an exception is generated if the result is invalid.

Figure 7. SafeInt solution

```
1. //addition
2. SafeInt<T> operator +(SafeInt<T> rhs) {
3.     return SafeInt<T>(addition(m_int, rhs.Value()));
4. }
5. int main(int argc, char *const *argv) {
6.     try {
7.         SafeInt<unsigned long> s1(strlen(argv[1]));
8.         SafeInt<unsigned long> s2(strlen(argv[2]));
9.         char *buff = (char *) malloc(s1 + s2 + 1);
10.        strcpy(buff, argv[1]);
11.        strcat(buff, argv[2]);
12.    }
13.    catch(SafeIntException err) {
14.        abort();
15.    }
16. }
```

The SafeInt library has several advantages over the Howard approach. Because it is written entirely in C++, it is more portable than safe arithmetic operations that depend on assembly language instructions. It is also more usable: arithmetic operators can be used in normal inline expressions, and SafeInt uses C++ exception handling instead of C-style return code checking. Performance of the library depends on many variables, but is generally better than the Howard approach when the optimizer is enabled.

The precondition approach could also be implemented in C-compatible libraries, although the advantages derived from C++ would not be realized.

References

[Howard 03b]

Howard, Michael. *An Overlooked Construct and an Integer Overflow Redux*.

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/>

(2003).

[LeBlanc 04]

LeBlanc, David. *Integer Handling with the C++ SafeInt Class*.

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/>
(2004).

Pearson Education, Inc. Copyright

This material is excerpted from *Secure Coding in C and C++*, by Robert C. Seacord, copyright © 2006 by Pearson Education, Inc., published as a CERT® book in the SEI Series in Software Engineering. All rights reserved. It is reprinted with permission and may not be further reproduced or distributed without the prior written consent of Pearson Education, Inc.

Fields

Name	Value
Copyright Holder	Pearson Education

Fields

Name	Value
is-content-area-overview	false
Content Areas	Knowledge/Coding Practices
SDLC Relevance	Implementation
Workflow State	Publishable